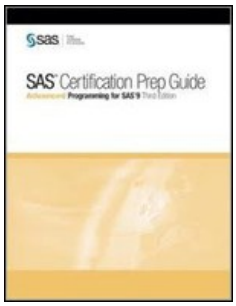


SAS Certification Prep Guide: Advanced Programming for SAS 9, Third Edition

by SAS Institute

Chapter 11: Creating and Using Macro Programs

Overview

Introduction

Like macro variables, macro programs (also known as macros) enable you to *substitute text* into your SAS programs. Macros are different from macro variables because they can use conditional logic to make decisions about the text that you substitute into your programs. Using macros can help make your SAS programs more dynamic and reusable.

For example, suppose you submit a SAS program every day to create registration listings for courses that are to be held later in the current month. Then, suppose that every Friday you also submit a SAS program to create a summary of revenue that has been generated so far in the current month. By using a macro, you can automate the process so that only one SAS program is required. This program will always submit the daily report and will conditionally submit the weekly report if it is Friday. Furthermore, you could create and store a macro that would automate this process, and the only code you would need to submit each day is this:

```
%reports
```

Objectives

In this chapter, you learn to

- define and call simple macros
- describe the basic actions that the macro processor performs during macro compilation and execution
- use system options for macro debugging
- interpret error messages and warning messages that the macro processor generates
- define and call macros that include parameters
- describe the difference between positional parameters and keyword parameters
- explain the difference between the global symbol table and local symbol tables
- describe how the macro processor determines which symbol table to use
- describe the concept of nested macros and the hierarchy of symbol tables
- conditionally process code within a macro program
- iteratively process code within a macro program.

Prerequisites

Before beginning this chapter, you should complete the following chapters:

- "Introducing Macro Variables" on page 304
- "Processing Macro Variables at Execution Time" on page 344.

Basic Concepts

Defining a Macro

In order to create a macro program, you must first define it. You begin a macro definition with a %MACRO statement, and you end the definition with a %MEND statement.

General form, %MACRO statement and %MEND statement:

%MACRO *macro-name*;

text

%MEND <*macro-name*>;

where

macro-name

names the macro. The value *macro-name* can be any valid SAS name that is not a reserved word in the SAS macro facility.

text

can be

- constant text, possibly including SAS data set names, SAS variable names, or SAS statements
- macro variables, macro functions, or macro program statements
- any combination of the above.

Tip You might want to include *macro-name* in the %MEND statement in order to make your program more readable. However, the inclusion *macro-name* in the %MEND statement is entirely optional.

Example

This program creates a macro named *Prtlast* that will print the most recently created data set. (Remember that the automatic macro variable **SYSLAST** stores the name of the most recently created data set.)

```
%macro prtlast;
   proc print data=&syslast (obs=5);
   title "Listing of &syslast data set";
   run;
%mend;
```

Compiling a Macro

In order to use this macro later in your SAS programs, you must first *compile* it by

```
%macro prtlast;
   proc print data=&syslast (obs=5);
   title "Listing of &syslast data set";
   run;
%mend;
```

When you submit this code, the word scanner divides the macro into tokens and sends the tokens to the macro processor for compilation. The macro processor

- checks all macro language statements for syntax errors (non-macro language statements are not checked until the macro is executed).
- writes error messages to the SAS log and creates a *dummy (non-executable) macro* if any syntax errors are found in the macro language statements.
- stores all compiled macro language statements and constant text in a SAS catalog entry if no syntax errors are found in the macro language statements. By default, a catalog named *Work.Sasmacr* is opened, and a catalog entry named *Macro-name.Macro* is created.

That is, if there are no syntax errors in the macro language statements within the macro, the text between the %MACRO statement and the %MEND statement will be stored under the name *Prtlast* for execution at a later time.

Note You can also store a compiled macro in a permanent SAS catalog. You can learn how to do this in "Storing Macro Programs" on page 442.

The MCOMPILENOTE= Option

The MCOMPILENOTE= option causes a note to be issued to the SAS log when a macro has completed compilation.

General form, MCOMPILENOTE= option:

OPTIONS MCOMPILENOTE= NONE | NOAUTOCALL | ALL;

where the option can take one of the three values listed and

NONE
: is the default value, which specifies that no notes are issued to the log.

NOAUTOCALL
: specifies that a note is issued to the log for completed macro compilations for all macros except autocall macros.

ALL
: specifies that a note is issued to the log for all completed macro compilations.

Note You can learn more about autocall macros in "Storing Macro Programs" on page 442.

Example

A macro might actually compile and still contain errors. If there are any errors, an ERROR message will be written to the SAS log in addition to the note. Here is an example of the note that is written to the log when a macro compiles without errors:

```
options mcompilenote=all;
%macro mymacro;
%mend mymacro;
```

Table 11.1: SAS Log

```
1. options mcompilenote=all;
2. %macro mymacro;
3. %mend mymacro;
NOTE: The macro MYMACRO completed compilation without errors.
```

Calling a Macro

After the macro is successfully compiled, you can use it in your SAS programs for the duration of your SAS session without resubmitting the macro definition. Just as you must reference macro variables in order to access them in your code, you must call a macro program in order to execute it within your SAS program.

A macro call

- is specified by placing a percent sign (%) before the name of the macro
- can be made anywhere in a program *except* within the data lines of a DATALINES statement (similar to a macro variable reference)
- requires *no semicolon* because it is *not* a SAS statement.

To execute the macro *Prtlast* you would call the macro as follows:

```
%prtlast
```

Caution A semicolon after a macro call might insert an in appropriate semicolon into the resulting program, leading to errors during compilation or execution.

Macros come in three types, depending on how they are called: name style, command style, and statement style. Of the three, name style is the most efficient. This is because calls to name style macros always begin with a percent sign (%), which immediately tells the word scanner to pass the token to the macro processor. With the other two types, the word scanner does not know immediately whether the token should be sent to the macro processor or not. Therefore, time is wasted while the word scanner determines this. All of the macros in this chapter are name style macros.

Example

Suppose a SAS program consists of several program steps that create SAS data sets. Suppose that after each of these program steps you want to print out the data set that has been created. Remember that the macro *Prtlast* prints the most recently created data set. If *Prtlast* has been compiled, you can call it after each step in order to print each data set.

```
proc sort data=sasuser.courses out=courses;
   by course_code;
run;

%prtlast

proc sort data=sasuser.schedule out=schedule;
   by begin_date;
run;

%prtlast

proc sort data=sasuser.students out=students;
   by student_name;
run;

%prtlast
```

Note The example above is simply meant to show you how you can incorporate a macro into your SAS program. Although this is a valid use of the *Prtlast* macro, this might not be the best way to code this example. Since the *Prtlast* macro uses no conditional logic or macro programming statements and it makes no decisions, this example does not illustrate the full power of a macro program. In the rest of this chapter, you will see examples of macro programs that are more useful than this one.

Macro Execution

When you call a macro in your SAS program, the word scanner passes the macro call to the macro processor, because the percent sign that precedes the macro name is a macro trigger. When the macro processor receives *%macro-name*, it

1. searches the designated SAS catalog *(Work.Sasmacr* by default) for an entry named *Macro-name.Macro*
2. executes compiled macro language statements within *Macro-name*
3. sends any remaining text in *Macro-name* to the input stack for word scanning
4. suspends macro execution when the SAS compiler receives a global SAS statement or when it encounters a SAS step boundary
5. resumes execution of macro language statements after the SAS code executes.

Later in this chapter you will see detailed examples of macro execution. These examples will make more sense once you have learned how to write a more complex macro program than you have seen so far in this chapter.

For now, remember that the macro call is processed by the macro processor *before* any SAS language statements such as DATA steps are compiled or executed. During macro execution, the macro processor can communicate directly with

- both global and local symbol tables. For example, the macro processor can store macro variable values with a %LET statement and can resolve macro variable references.
- the input stack. For example, the macro processor can generate SAS code for tokenization by the word scanner.

Note You will learn more about global and local symbol tables later in this chapter.

Example

This example demonstrates macro execution. Assume that the *Prtlast* macro has been compiled and that it has been stored in the *Work.Sasmacr* catalog.

1. First, you submit the macro call, as follows:

   ```
   %prtlast
   ```

2. When the word scanner encounters this call, it passes the call to the macro processor. The macro processor searches for the compiled macro in the catalog entry *Work.Sasmacr.Prtlast.Macro*.

   Catalog Entry

   ```
   %macro prtlast;
      proc print data=&syslast(obs=5);
      title "Listing of &syslast data set";
      run;
   %mend;
   ```

3. The macro processor begins executing compiled macro language statements. However, in this example, no compiled macro statements are included in the macro.

4. The macro processor places noncompiled items (SAS language statements) on the input stack, and pauses as the word scanner tokenizes the inserted text. In this example, the macro processor will place the PROC PRINT step on the input stack.

   Input Stack

   ```
   proc print data=&syslast(obs=5);
      title "Listing of &syslast data set";
   run;
   ```

5. The word scanner passes these tokens to the compiler. When the word scanner encounters a macro variable reference such as *&syslast*, it passes the reference to the macro processor for resolution. The macro processor returns the macro variable value to the input stack and word scanning continues.

6. After all of the statements in the PROC PRINT step have been compiled, the PROC PRINT step is executed, and SAS creates output that includes only the first five observations of the most recently created data set.

7. Once the PROC PRINT step has been executed, the macro processor resumes execution of any remaining macro language statements in the macro (there are none in this example). The macro processor ends execution when it reaches the %MEND statement.

Assume that the most recently created data set is *Work.Practice* (which is a copy of *Sasuser.Courses)*. Here is the output that is generated by calling the *Prtlast* macro.

Listing of WORK.PRACTICE date set

Obs	Course_Code	Course_Title	Days	Fee
1	C001	Basic Telecommunications	3	$795
2	C002	Structured Query Language	4	$1150
3	C003	Local Area Networks	3	$650
4	C004	Database Design	2	$375
5	C005	Artificial Intelligence	2	$400

Here is an example of messages that are written to the SAS log when **`%prtlast`** is submitted, assuming that the most recently created data set is *Work.Practice*.

Table 11.2: SAS Log

```
37    %prtlast

NOTE: Writing HTML Body file: sashtm3.htm
```

```
NOTE: There were 5 observations read from the data set
      WORK.PRACTICE.
NOTE: PROCEDURE PRINT used (Total process time):
      real time          0.04 seconds
      cpu time           0.03 seconds
```

Notice that in this SAS log message, you see a note from PROC PRINT, but not the PROC PRINT code it self since the call to the macro does not display the text that is sent to the compiler.

This example depicts the processing and execution of a simple macro. Later in this chapter you will see how more-complex macros that include compiled macro language statements are handled by the macro processor.

Developing and Debugging Macros

Monitoring Execution with System Options

In the last example, you saw that when you call a macro, the text that is sent to the compiler does not appear in the SAS log. But sometimes you might want to see this text. There are SAS system options that can display information about the execution of a macro in the SAS log. This can be especially helpful for debugging purposes.

The MPRINT Option

When the MPRINT option is specified, the text that is sent to the SAS compiler as a result of macro execution is printed in the SAS log.

General form, MPRINT | NOMPRINT option:

OPTIONS MPRINT | NOMPRINT;

where

NOMPRINT
: is the default setting, and specifies that the text that is sent to the compiler when a macro executes is not printed in the SAS log.

MPRINT
: specifies that the text that is sent to the compiler when a macro executes is printed in the SAS log.

You might want to specify the MPRINT system option if

- you have a SAS syntax error or execution error
- you want to see the generated SAS code.

The MPRINT system option is often synchronized with the SOURCE system option to show, or hide, executed SAS code.

Example

Suppose you want to call the *Prtlast* macro and to use the MPRINT system option to show the SAS code that results from the macro execution.

Catalog Entry

```
%macro prtlast;
   proc print data=&syslast (obs=5);
      title "Listing of &syslast data set";
   run;
%mend;
```

The following sample code creates a data set named *Sales*, specifies the MPRINT option, and references the *Prtlast* macro:

```
data sales;
   price_code=1;
run;
options mprint;
%prtlast
```

The messages that are written to the SAS log show the text that is sent to the compiler. Notice that the macro variable reference **(&SYSLAST)** is resolved to the value *Work.Sales* in the MPRINT messages that are written to the SAS log.

Table 11.3: SAS Log

```
101 %prtlast
MPRINT(PRTLAST): proc print data=WORK.SALES (obs=5);
MPRINT(PRTLAST): title "Listing of WORK.SALES";
MPRINT(PRTLAST): run;
NOTE: There were 1 observations read from the dataset WORK.SALES.
NOTE: PROCEDURE PRINT used:
      real time           0.04 seconds
      cpu time            0.04 seconds
```

The MLOGIC Option

Another system option that might be useful when you debug your programs is the MLOGIC option. The MLOGIC option prints messages that indicate macro actions that were taken during macro execution.

General form, MLOGIC| NOMLOGIC option:

OPTIONS MLOGIC | NOMLOGIC;

where

NOMLOGIC
: is the default setting, and specifies that messages about macro actions are not printed to the SAS log during macro execution.

MLOGIC
: specifies that messages about macro actions are printed to the log during macro execution.

When the MLOGIC system option is in effect, the information that is displayed in SAS log messages includes

- the beginning of macro execution
- the values of macro parameters at invocation
- the execution of each macro program statement
- whether each %IF condition is true or false
- the end of macro execution.

Example

Suppose you want to repeat the previous example with only the MLOGIC system option in effect. This sample code creates a data set named *Sales*, sets the MLOGIC system option, and calls the *Prtlast* macro.

```
data sales;
   price_code=1;
run;
options nomprint mlogic;
%prtlast
```

When this code is submitted, the messages that are written to the SAS log show the beginning and the end of macro

processing.

Table 11.4: SAS Log

```
107    %prtlast
MLOGIC (PRTLAST): Beginning execution.
NOTE:  There were 1 observations read from the dataset WORK.SALES.
NOTE:  PROCEDURE PRINT used:
       real time           0.02 seconds
       cpu time            0.02 seconds
MLOGIC(PRTLAST): Ending execution.
```

The MLOGIC option, along with the SYMBOLGEN option, is typically turned

- *on* for development and debugging purposes
- *off* when the application is in production mode.

Comments in Macro Programs

As with any other programs, your macro programs might benefit from comments. Comments can be especially helpful if you plan to save your macros permanently or to share them with other users. You can place comments within a macro definition by using the macro comment statement.

General form, macro comment statement:

*%*comment;*

where

comment

can be any message. Like other SAS statements, each macro comment statement ends with a semicolon.

Example

The following code uses macro comments to describe the functionality of the macro:

```
%* The value of &syslast will be substituted appropriately ;
%* as long as a data set has been created during this session. ;
   proc print data=&syslast(obs=5);
/* Print only the first 5 observations */
   title "Last Created Data Set Is &syslast";
   run;
%mend;
```

Note You can also use the comment symbols /* and */inside a macro. When these symbols appear, the macro processor ignores the text within the comment.

Using Macro Parameters

You have seen the basic form for a macro definition. Your macros will often contain macro variables. To make your macros more dynamic, you could use the %LET statement to update the values of the macro variables that are used within the macros. However, *parameter lists* in your macro definitions enable you to update the macro variables within your macro programs more conveniently. A parameter list is an optional part of the %MACRO statement that names one or more macro variables whose values you specify when you call the macro.

Example

Suppose the compiled macro *Printdsn* contains references to the macro variables **`dsn`** (which records a data set name) and **`vars`** (which records a list of data set variables), as follows:

```
%macro printdsn;
   proc print data=&dsn;
      var &vars;
   title "Listing of %upcase(&dsn) data set";
   run;
%mend;
```

You could modify the behavior of *Printdsn* by changing the value of macro variable **dsn** or **vars** with a %LET statement before you call the macro. For example, you could substitute *sasuser.courses* for **dsn** and *course_codecourse_title days* for **vars** at macro execution, as follows:

```
%let dsn=sasuser.courses;
%let vars=course_code course_title days;
%printdsn
```

If the MPRINT system option is turned on when this code is submitted, the following messages are written to the SAS log. Notice that the values that you provided in the %LET statements have been substituted into the macro when it appears in the SAS log. Then you could submit new %LET statements in order to change the value of **dsn** to *sasuser.schedule* and to change the value of **vars** to *course_code location begin_date* when the macro executes, as follows:

Table 11.5: SAS Log

```
7      options mprint;
8      %let dsn=sasuser.courses;
9      %let vars=course_code course_title days;
10     %printdsn
NOTE: Writing HTML Body file: sashtm.htm
MPRINT(PRINTDSN): proc print data=sasuser.courses;
MPRINT(PRINTDSN): var course_codecourse_title days;
MPRINT(PRINTDSN): title "Listing of SASUSER.COURSES data set";
MPRINT(PRINTDSN): run;
NOTE: There were 6 observations read from the data set
      SASUSER.COURSES.
NOTE: PROCEDURE PRINT used (Total process time):
      real time          6.59 seconds
      cpu time           0.28 seconds
```

```
%let dsn=sasuser.schedule;
%let vars=course_code location begin_date;
%printdsn
```

The messages that are written to the SAS log when this code is submitted show that the new values have been substituted for the macro variable references in the macro.

Table 11.6: SAS Log

```
11    %let dsn=sasuser.schedule;
12    %let vars=course_code location begin_date;
13    %printdsn
MPRINT(PRINTDSN):   proc print data=sasuser.schedule;
MPRINT(PRINTDSN):   var course_code location begin_date;
MPRINT(PRINTDSN):   title "Listing of SASUSER.SCHEDULE data set";
MPRINT(PRINTDSN):   run;
NOTE: Writing HTML Body file: sashtm1.htm
NOTE: There were 18 observations read from the data set
      SASUSER.SCHEDULE.
NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.76 seconds
      cpu time            0.08 seconds
```

You can make these macro variables easier to update by using parameters in the macro definition to create the macro variables. Then you can pass values to the macro variables each time you call the macro rather than using separate %LET statements. The next few sections will show you how to use various types of parameters to create macro variables.

Macros That Include Positional Parameters

When you include positional parameters in a macro definition, a macro variable is automatically created for each parameter when you call the macro. To define macros that includ *Q positional parameters*, you list the names of macro variables in the %MACRO statement of the macro definition. Positional parameters are so named because the order in which you specify them in a macro definition determines the order in which they are assigned values from the macro call. That is, when you call a macro that includes positional parameters, you specify the values of the macro variables that are defined in the parameters in the *same order* in which they are defined.

General form, macro definition that includes positional parameters:

%MACRO *macro-name(parameter-l<,…,parameter-n>*);

text

%MEND *<macro-name>;*

where

parameter-l<,…,parameter-n>

specifies one or more positional parameters, separated by commas. You must supply each parameter with a name: you cannot use a text expression to generate it.

To *call* a macro that includes positional parameters, precede the name of the macro with a percent sign, and enclose the parameter values in parentheses. List the values in the same order in which the parameters are listed in the macro definition, and separate them with commas, as follows:

%macro-name(value-l<,…,value-n>)

The values listed in a macro call

- can be null values, text, macro variable references, or macro calls
- are assigned to the parameter variables using a one-to-one correspondence.

Example

You can use positional parameters to create the macro variables **dsn** and **vars** in the *Printdsn* macro definition, as follows:

```
%macro printdsn(dsn,vars);
   proc print data=&dsn;
      var &vars;
   title "Listing of %upcase(&dsn) data set";
   run;
%mend;
```

In this case, when you call the *Printdsn* macro you assign values to the macro variables that are created in the parameters. In the following example, the value *sasuser.courses* is assigned to the macro variable **dsn**, and the value *course_codecourse_title days* is assigned to the macro variable **vars**. Notice that the value for **dsn** is listed first and the value for **vars** is listed second, since this is the order in which they are listed in the macro definition.

```
%printdsn(sasuser.courses,course_code course_title days)
```

Note To substitute a null value for one or more positional parameters, use commas as placeholders for the omitted values, as follows:

```
%printdsn(,course_code course_title days)
```

Macros That Include Keyword Parameters

You can also include *keyword parameters* in a macro definition. Like positional parameters, keyword parameters create macro variables. However, when you use keyword parameters to create macro variables, you specify the name, followed by the equals sign, and the value of each macro variable in the macro definition.

Keyword parameters can be listed in any order. Whatever value you assign to each parameter (or variable) in the %MACRO statement becomes its default value. Null values are allowed.

General form, macro definition that includes keyword parameters:

%MACRO *macro-name(keyword-l=<value-l><,…,keyword-n=<value-n>>*);

text

%MEND <*macro-name>;*

where

keyword-l=<value-l><,…,keyword-n=<value-n>>

names one or more macro parameters followed by equal signs. You can specify default values after the equal signs. If you omit a default value, the keyword parameter has a null value.

When you *call* a macro whose definition includes keyword parameters, you specify the keyword, followed by the equals sign, and the value for each parameter, in any order. If you omit a keyword parameter from the macro call, the keyword variable retains its default value, as follows:

```
%macro-name(keyword-1=value-1<,…,keyword-n=value-n>)
```

Example

You can use keyword parameters to create the macro variables **dsn** and **vars** in the *Printdsn* macro. This example assigns a default value of *sasuser.courses* to the macro variable **dsn** and assigns a default value of *course_codecourse_title days* to the macro variable **vars:**

```
%macro printdsn(dsn=sasuser.courses,
                vars=
course_codecourse_title days);
   proc print data=&dsn;
      var &vars;
   title "Listing of %upcase(&dsn) data set";
   run;
%mend;
```

To invoke the *Printdsn* macro with a value of *sasuser.schedule* for **dsn** and a value of *teacher course_titlebegin_date* for **vars**, issue the following call:

```
%printdsn(dsn=sasuser.schedule, vars=teacher course_code begin_date)
```

To call the *Printdsn* macro with default values for the parameters *(sasuser.courses* as the value for **dsn** and *course_codecourse_title days* as the value for **vars)**, you could issue the following call:

```
%printdsn()
```

Note To call the macro *Printdsn* with default values for the parameters, you could also issue a macro call that specified these values explicitly, as follows:

```
%printdsn(dsn=sasuser.courses,vars=course_code course_title days)
```

Macros That Include Mixed Parameter Lists

You can also include a parameter list that contains *both positional and keyword* parameters in your macro definitions. All positional parameter variables in the %MACRO statement must be listed *before* any keyword parameter variable is listed.

General form, macro definition that includes mixed parameters:

%MACRO *macro-name(parameter-l<,…,parameter-n>*,

keyword-l=<value-l><,…,keyword-n=<value-n>>);

text

%MEND;

where

parameter-l<,…,parameter-n>

is listed before *keyword-l = <value-l><,…,keyword-n=<value-n>>*.

Similarly, when you *call* a macro that includes a mixed parameter list, you must list the positional values before any keyword values, as follows:

```
%macro-name(value-1<,…,value-n>,
            keyword-1=value-1<,…,keyword-n=value-n>)
```

Example

You can use a combination of positional and keyword parameters to create the macro variables in the *Printdsn* macro definition. This code uses a positional parameter to create the macro variable **dsn**, and a keyword parameter to create the macro variable **vars:**

```
%macro printdsn(dsn, vars=course_titlecourse_code days);
     proc print data=&dsn;
        var &vars;
     title "Listing of %upcase(&dsn) data set";
     run;
%mend;
```

The following call to the *Printdsn* macro assigns the value *sasuser.schedule* to the macro variable **dsn** and assigns the value *teacher location begin_date* to the macro variable **vars**. Notice that the value for **dsn** is listed first, since **dsn** is the positional parameter.

```
%printdsn(sasuser.schedule, vars=teacher location begin_date)
```

Now, suppose you want to execute the *Printdsn* macro, assigning the default value *course_titlecourse_code days* to the macro variable **vars** and assigning the value *sasuser.courses* to the macro variable **dsn**. You could issue the following call:

```
%printdsn(sasuser.courses)
```

Because this call omits the keyword parameter **(vars)**, the default value for that parameter is used.

Macros That Include the PARMBUFF Option

You can use the PARMBUFF option in a macro definition to create a macro that can accept a *-varying number of parameters* at each invocation. The PARMBUFF option assigns the entire list of parameter values in a macro call, including the parentheses in a name-style invocation, as the value of the automatic macro variable **SYSPBUFF**.

General form, macro definition with the PARMBUFF option:

%MACRO *macro-name* **/PARMBUFF;**

text

%MEND;

where

text

contains a reference to the automatic macro variable **SYSPBUFF**.

Example

The following macro definition creates a macro named *Printz. Printz* uses a varying number of parameters and the automatic macro variable **SYSPBUFF** to display the parameters that are specified in the macro call. The macro also uses a loop to print the data sets that are named as parameters.

```
%macro printz/parmbuff;
   %put Syspbuff contains: &syspbuff;
   %local num;
   %do num=1 %to %sysfunc(countw(&syspbuff));
   %let dsname=%scan(&syspbuff,&num);
      proc print data=sasuser.&dsname;
      run;
   %end;
%mend printz;
```

If you submit a call to the macro that includes two parameters, the *Printz* macro writes the following line to the SAS log and causes two data sets to be printed

```
%printz(courses, schedule)
```

Table 11.7: SAS Log

```
Syspbuff contains: (courses,schedule)
```

If you submit a call to the macro that includes one parameter, the *Printz* macro writes the following line to the SAS log and causes one data set to be printed:

```
%printz(courses)
```

Table 11.8: SAS Log

```
Syspbuff contains: (courses)
```

Note If the macro definition includes both a set of parameters and the PARMBUFF option, the macro invocation causes the parameters to receive values and the entire invocation list of values to be assigned to **SYSPBUFF**.

Understanding Symbol Tables

The Global Symbol Table

You are already somewhat familiar with the *global symbol table*. Remember that automatic macro variables are stored in the global symbol table. User-defined macro variables that you create with a %LET statement in open code (code that is outside of a macro definition) are also stored in the global symbol table.

Global Symbol Table	
SYSDATE	*04APR11*
SYSDAY	*Monday*
SYSVER	*9.2*
uservarl	*value1*
uservar2	*value2*

The global symbol table is created during the initialization of a SAS session and is deleted at the end of the session. Macro variables in the global symbol table

- are available anytime during the session
- can be created by a user
- have values that can be changed during the session (except for some automatic macro variables).

You can create a global macro variable with

- a %LET statement (used outside a macro definition)
- a DATA step that contains a SYMPUT routine
- a DATA step that contains a SYMPUTX routine
- a SELECT statement that contains an INTO clause in PROC SQL
- a %GLOBAL statement.

You should already be familiar with the %LET statement, the SYMPUT routine, and the INTO clause. We will now examine the %GLOBAL statement.

The %GLOBAL Statement

The %GLOBAL statement

- creates one or more macro variables in the global symbol table and assigns null values to them
- can be used either inside *or* outside a macro definition
- has no effect on variables that are already in the global symbol table.

General form, %GLOBAL statement:

%GLOBAL *macro-variable-1 <…macro-variable-n>;*

where

macro-variable

is either the name of a macro variable or a text expression that generates a macro variable name.

Example

To create a global macro variable inside a macro definition, you can use the %GLOBAL statement. The %GLOBAL statement in the following example creates two global macro variables, **dsn** and **vars**. The %LET statements assign values to the new global macro variables, as follows:

```
%macro printdsn;
   %global dsn vars;
   %let dsn=sasuser.courses;
   %let vars=course_titlecourse_code days;
   proc print data=&dsn;
      var &vars;
   title "Listing of &dsn data set";
   run;
%mend;

%printdsn
```

Note You use the *%SYMDEL statement* to delete a macro variable from the global symbol table during a SAS session. To remove the macro variable **dsn** from the global symbol table, you submit the following statement:

```
%symdel dsn;
```

The Local Symbol Table

A *local symbol table* is created when a macro that includes a parameter list is called or when a request is made to create a local variable during macro execution. The local symbol table is deleted when the macro finishes execution. That is, the local symbol table exists only while the macro executes.

Local Symbol Table	
parameterl	*value1*
parameter2	*value2*
uservar1	*value1*
uservar2	*value2*

The local symbol table contains macro variables that can be

- created and initialized at macro invocation (that is, by parameters)
- created or updated during macro execution
- referenced anywhere within the macro.

You can create local macro variables with

- parameters in a macro definition
- a %LET statement within a macro definition
- a DATA step that contains a SYMPUT routine within a macro definition
- a DATA step that contains a SYMPUTX routine within a macro definition
- a SELECT statement that contains an INTO clause in PROC SQL within a macro definition
- a %LOCAL statement.

Note The SYMPUT routine can create a local macro variable if a local symbol table already exists. If no local symbol table exists when the SYMPUT routine executes, it will create a global macro variable.

You have already learned about using parameters in macro definitions. You should also already be familiar with the %LET statement, the SYMPUT routine, and the INTO clause. We will examine the %LOCAL statement.

The %LOCAL Statement

The %LOCAL statement

- can appear *only* inside a macro definition
- creates one or more macro variables in the local symbol table and assigns null values to them
- has no effect on variables that are already in the local symbol table.

A local symbol table is not created until a request is made to create a local variable. Macros that do not create local variables do not have a local table. Remember, the SYMPUT routine can create local variables only if the local table already exists.

Since local symbol tables exist separately from the global symbol table, it is possible to have a local macro variable and a global macro variable that have the same name and different values.

Example

In this example, the first %LET statement creates a global macro variable named **dsn** and assigns a value of *sasuser.courses* to it.

The %LOCAL statement within the macro definition creates a local macro variable named **dsn**, and the %LET statement within the macro definition assigns a value of *sasuser.register* to the local variable **dsn**.

The %PUT statement within the macro definition will write the value of the local variable **dsn** to the SAS log, whereas the %PUT statement that follows the macro definition will write the value of the global variable **dsn** to the SAS log:

```
%let dsn=sasuser.courses;
```

```
%macro printdsn;
   %local dsn;
   %let dsn=sasuser.register;
   %put The value of DSN inside Printdsn is &dsn;
%mend;

%printdsn
%put The value of DSN outside Printdsn is &dsn;
```

When you submit this code, the following statements are written to the SAS log.

Table 11.9: SAS Log

```
199  %let dsn=sasuser.courses;
200
201  %macro printdsn;
202     %local dsn;
203     %let dsn=sasuser.register;
204     %put The value of DSN inside Printdsn is &dsn;
205  %mend;
206
207  %printdsn
The value of DSN inside Printdsn is sasuser.register
208  %put The value of DSN outside Printdsn is &dsn;
The value of DSN outside Printdsn is sasuser.courses
```

Rules for Creating and Updating Variables

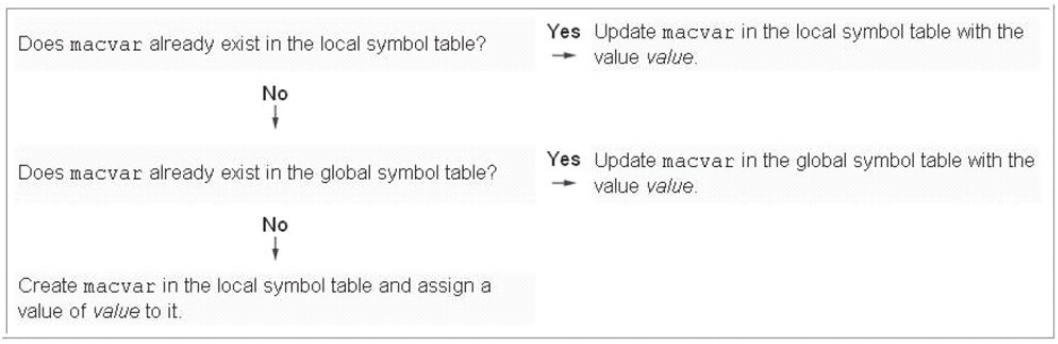
When the macro processor receives a request to create or update a macro variable during

macro execution, the macro processor follows certain rules. Suppose the macro processor receives a %LET statement during a macro call, as follows:

```
%let macvar=value;
```

The macro processor will take the following steps:

1. The macro processor checks to see whether the macro variable **`macvar`** already exists in the local symbol table. If so, the macro processor updates **`macvar`** in the local symbol table with the value *value*. If **`macvar`** does not exist in the local table, the macro processor goes on to step 2.
2. The macro processor checks to see whether the macro variable **`macvar`** already exists in the global symbol table. If so, the macro processor updates **`macvar`** in the global symbol table with the value *value*. If **`macvar`** does not exist in the global symbol table, the macro processor goes on to step 3.
3. The macro processor creates a macro variable named **`macvar`** in the local symbol table and assigns a value of *value* to it.

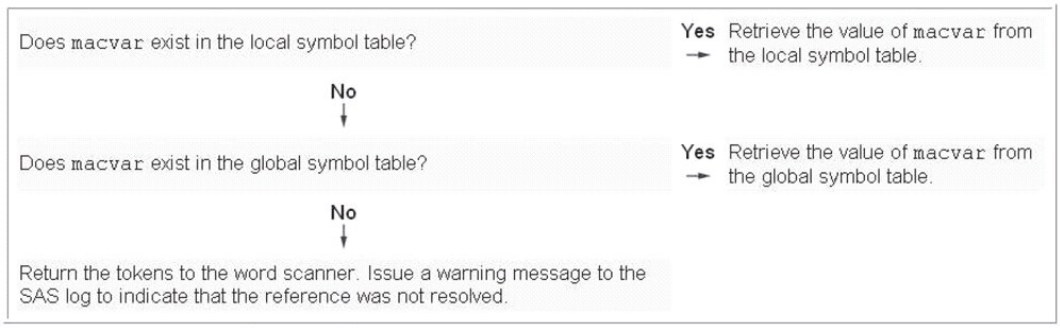


Similarly, suppose the macro processor receives the following reference during a macro call:

```
&macvar
```

The macro processor will take the following steps:

1. The macro processor checks to see whether the macro variable **macvar** exists in the local symbol table. If so, the macro processor retrieves the value of **macvar** from the local symbol table. If **macvar** does not exist in the local table, the macro processor goes on to step 2.
2. The macro processor checks to see whether the macro variable **macvar** exists in the global symbol table. If so, the macro processor retrieves the value of **macvar** from the global symbol table. If **macvar** does not exist in the global symbol table, the macro processor goes on to step 3.
3. The macro processor returns the tokens to the word scanner. A warning message is written to the SAS log to indicate that the reference was not resolved.



Note Remember that if the macro processor receives either a %LET statement or a macro variable reference **(&macvar)** in open code, it will check *only* the *global* symbol table for existence of the macro variable. If a macro program is not currently executing, a local symbol table does not currently exist.

Multiple Local Symbol Tables

Multiple local symbol tables can exist *concurrently* during macro execution if you have *nested macros*. That is, if you define a macro program that calls another macro program, and if both macros create local symbol tables, then two local symbol tables will exist while the second macro executes.

Example

Suppose the following two macros, *Outer* and *Inner*, have been compiled. The macro named *Outer* creates a local macro variable named **variX** and assigns a value of *one* to it. Then *Outer* calls another macro program named *Inner*. The macro named *Inner* creates a local macro variable named **variY** and assigns the value of *variX* to it.

```
%macro outer;
   %local variX;
   %let variX=one;
   %inner
%mend outer;

%macro inner;
   %local variY;
   %let variY=&variX;
%mend inner;
```

We will examine what happens to the symbol tables when you submit the following code:

```
%let variX=zero;
%outer
```

1. The macro processor receives **%let variX=zero;**. It checks the global symbol table for a macro variable named **variX**. There is none, so the macro processor creates **variX** and assigns a value of *zero* to it.

Global Symbol Table	
VariY	*Zero*

2. The macro processor receives **`%outer`**. The macro processor retrieves the macro *Outer* from *Work.Sasmacr*, and begins executing it.

3. The macro processor encounters **`%local variX;`**. It creates a local symbol table. The macro processor creates the macro variable **`variX`** in this local table and assigns a null value to it. This does not affect the macro variable **`variX`** that is stored in the global symbol table.

Global Symbol Table	
VariX	*Zero*

Outer Local Symbol Table	
VariX	

4. The macro processor encounters **`%let variX=one;`**. The macro processor checks the local symbol table for **`variX`** and assigns a value of *one* to it.

Global Symbol Table	
VariX	*Zero*

Outer Local Symbol Table	
VariY	*One*

5. The macro processor receives **`%inner`**. It retrieves the macro *Inner* from *Work.Sasmacr*, and begins executing it.

6. The macro processor encounters **`%local variY;`**. It creates a local symbol table. The macro processor creates a macro variable **`variY`** in this table and assigns a null value to it. There are now two local symbol tables in existence.

Global Symbol Table	
VariX	*Zero*

Outer Local Symbol Table	
VariX	*One*

Inner Local Symbol Table	
VariX	

7. The macro processor encounters **`%let variY=&variX;`**. It checks the most recently created local table for **`variX`**. There is no such macro variable in that symbol table, so the macro processor then checks the other local symbol table. It retrieves the value *one* from that symbol table and substitutes the value into the %LET statement. Then the macro processor checks the most recently created local symbol table for a macro variable named **`variY`**. When it finds this macro variable, it assigns the value *one* to it.

Global Symbol Table	
VariX	*Zero*

Outer Local Symbol Table	
VariX	*One*

Inner Local Symbol Table	
VariX	*One*

8. The *Inner* macro finishes executing, and the local symbol table that was created within this macro is deleted. There is

now only one local symbol table in existence.

Global Symbol Table	
VariX	*Zero*

Outer Local Symbol Table	
VariX	*One*

9. The *Outer* macro finishes executing, and the local symbol table that was created within this macro is deleted. There are now no local symbol tables in existence. The global symbol table has not been changed since **`variX`** was created and was assigned a value of *zero*.

Global Symbol Table	
VariX	*Zero*

As you can see, each macro program in the example above has its own local symbol table that exists as long as the macro executes. When a macro finishes executing, its local symbol table and all of the local macro variables that are contained in that table are erased. The global symbol table and all of the global macro variables that are contained in it remain.

The MPRINTNEST Option

The MPRINTNEST option allows the macro nesting information to be written to the SAS log in the MPRINT output. This has no effect on the MPRINT output that is sent to an external file.

General form, MPRINTNEST option:

OPTIONS MPRINTNEST | NOMPRINTNEST;

where

MPRINTNEST

specifies that macro nesting information is written in the MPRINT output in the SAS log.

NOMPRINTNEST

specifies that macro nesting information is not written in the MPRINT output in the SAS log.

The setting of the MPRINTNEST option does not imply the setting of MPRINT. You must set both MPRINT and MPRINTNEST in order for output with the nesting information to be written to the SAS log.

Example

Suppose that you have defined three nested macros, as follows:

```
%macro outer;
   data _null_;
      %inner
   run;
%mend outer;

%macro inner;
   put %inrmost;
%mend inner;


%macro inrmost;
   'This is the text of the PUT statement'
%mend inrmost;
```

The SAS log below shows the messages that are written when you set both the MPRINT and MPRINTNEST options and submit a call to the *Outer* macro, as follows:

```
options mprint mprintnest;
%outer
```

Table 11.10: SAS Log

```
MPRINT(OUTER):    data _null_;
MPRINT(OUTER.INNER): put
MPRINT(OUTER.INNER.INRMOST):  'This is the text of the PUT statement'
MPRINT(OUTER.INNER): ;
MPRINT(OUTER):   run;

This is the text of the PUT statement
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
```

The MLOGICNEST Option

The MLOGICNEST option allows the macro nesting information to be displayed in the MLOGIC output in the SAS log. The setting of MLOGICNEST does not affect the output of any currently executing macro.

General form, MLOGICNEST option:

OPTIONS MLOGICNEST | NOMLOGICNEST;

where

MLOGICNEST

specifies that macro nesting information is written in the MLOGIC output in the SAS log.

NOMLOGICNEST

specifies that macro nesting information is not written in the MLOGIC output in the SAS log.

The setting of MLOGICNEST does not imply the setting of MLOGIC. You must set both MLOGIC and MLOGICNEST in order for output with nesting information to be written to the SAS log.

Example

Suppose that you have defined three nested macros, as follows:

```
%macro outer;
   %put THIS IS OUTER;
   %inner
%mend outer;

%macro inner;
   %put THIS IS INNER;
   %inrmost
%mend inner;

%macro inrmost;
   %put THIS IS INRMOST;
%mend inrmost;
```

The SAS log below shows the messages that are written when you set both the MLOGIC and MLOGICNEST options and submit a call to the *Outer* macro, as follows:

```
options mlogic mlogicnest;
%outer
```

Table 11.11: SAS Log

```
MLOGIC(OUTER): Beginning execution.
MLOGIC(OUTER): %PUT THIS IS OUTER
THIS IS OUTER
MLOGIC(OUTER.INNER): Beginning execution.
MLOGIC(OUTER.INNER): %PUT THIS IS INNER
THIS IS INNER
MLOGIC(OUTER.INNER.INRMOST): Beginning execution.
MLOGIC(OUTER.INNER.INRMOST): %PUT THIS IS INRMOST
THIS IS INRMOST
MLOGIC(OUTER.INNER.INRMOST): Ending execution.
MLOGIC(OUTER.INNER): Ending execution.
MLOGIC(OUTER): Ending execution.
```

Processing Statements Conditionally

Conditional Execution

You can use macros to control conditional execution of statements. Remember the example from the beginning of this chapter where you wanted to run a daily report to create registration listings for courses to be held later in the month. Remember that you also wanted to run a weekly report each Friday to create a summary of revenue that has been generated so far in the current month. You can accomplish these tasks with one program if you use conditional execution to determine whether the second report should be run.

You can perform conditional execution at the macro level with %IF-%THEN and %ELSE statements.

General form, %IF-%THEN and %ELSE statements:

%IF *expression* **%THEN** *text;*

<%ELSE *text;*>

where

expression

can be any valid macro expression that resolves to an integer.

text

can be specified as

- constant text
- a text expression
- a macro variable reference, a macro call, or a macro program statement.

If *expression* resolves to zero, then it is false and the %THEN text is not processed (the optional %ELSE text is processed instead). If it resolves to any integer other than zero, then the expression is true and the %THEN text is processed. If it resolves to null or to any noninteger value, an error message is issued.

The %ELSE statement is optional. However, the macro language does not contain a subsetting %IF statement. Thus, you cannot use %IF without %THEN.

%IF-%THEN Compared to IF-THEN

Although they look similar, the %IF-%THEN/%ELSE statement and the IF-THEN/ELSE statement belong to two different languages. Most of the same rules that apply to the DATA step IF-THEN/ELSE statement also apply to the %IF-%THEN/%ELSE statement. However, there are several important differences between the macro %IF-%THEN statement and the

DATA step IF-THEN statement.

%IF-%THEN…	IF-THEN…
is used only in a macro program.	is used only in a DATA step program.
executes during macro execution.	executes during DATA step execution.
uses macro variables in logical expressions and cannot refer to DATA step variables in logical expressions.	uses DATA step variables in logical expressions.
determines what text should be copied to the input stack.	determines what DATA step statement(s) should be executed. When inside a macro definition, it is copied to the input stack as text.

Simple %DO and %END statements often appear in conjunction with %IF-%THEN/%ELSE statements in order to designate a section of the macro to be processed depending on whether the %IF condition is true or false. Use %DO and %END statements following %THEN or %ELSE in order to conditionally place text that contains multiple statements onto the input stack. Each %DO statement must be paired with an %END statement.

General form, %DO-%END with %IF-%THEN and %ELSE statements:

```
%IF expression %THEN %DO;
          text and/or macro language statements
%END;
%ELSE %DO;
          text and/or macro language statements
%END;
```

where

text and/or macro language statements

is either constant text, a text expression, and/or a macro statement.

Note The statements %IF-%THEN, %ELSE, %DO, and %END are macro language statements that can be used only inside a macro program.

Example

You can control text that is copied to the input stack with the %IF-%THEN while controlling DATA step logic with IF-THEN. In this example, the value of the macro variable **status** determines which variables will be included in the new data set. The value of the data set variable **Location** determines the value of the new data set variable **Totalfee**.

```
%macro choice(status);
   data fees;
      set sasuser.all;
      %if &status=PAID %then %do;
          where paid='Y';
          keep student_name course_code begin_date totalfee;
      %end;
      %else %do;
          where paid='N';
          keep student_name course_code
               begin_date totalfee latechg;
          latechg=fee*.10;
      %end;
      /* add local surcharge */
      if location='Boston' then totalfee=fee*1.06;
      else if location='Seattle' then totalfee=fee*1.025;
      else if location='Dallas' then totalfee=fee*1.05;
run;
%mend choice;
```

If the MPRINT and MLOGIC system options are both set, the SAS log will display messages showing the text that is sent to the compiler. For example, suppose you submit the following macro call:

```
options mprint mlogic;
%choice(PAID)
```

The following messages will be written to the log. Notice that the MLOGIC option shows the evaluation of the expression in the %IF statement, but it does not show the evaluation of the expression in the IF statement.

Table 11.12: SAS Log

```
160 %choice(PAID)
MLOGIC(CHOICE): Beginning execution.
MLOGIC(CHOICE): Parameter STATUS has value PAID
MPRINT(CHOICE): data fees;
MPRINT(CHOICE): set sasuser.all;
MLOGIC(CHOICE): %IF condition &status=PAID is TRUE
MPRINT(CHOICE): where paid='Y';
MPRINT(CHOICE): keep student_name course_code begin_date totalfee;
MPRINT(CHOICE): if location='Boston' then totalfee=fee*1.06;
MPRINT(CHOICE): else if location='Seattle' then totalfee=fee*1.025;
MPRINT(CHOICE): else if location='Dallas' then totalfee=fee*1.05;
MPRINT(CHOICE): run;
```

Suppose you submit the following macro call:

```
options mprint mlogic;
%choice(OWED)
```

The following messages will be sent to the SAS log. Notice that the text that is written to the input stack is different this time.

Table 11.13: SAS Log

```
161 %choice(OWED)
MLOGIC(CHOICE): Beginning execution.
MLOGIC(CHOICE): Parameter STATUS has value OWED
MPRINT(CHOICE): data fees;
MPRINT(CHOICE): set sasuser.all;
MLOGIC(CHOICE): %IF condition &status=PAID is FALSE
MPRINT(CHOICE): where paid='N';
MPRINT(CHOICE): keep student_namecourse_codebegin_date totalfee
                latechg;
MPRINT(CHOICE): latechg=fee*.10;
MPRINT(CHOICE): if location='Boston' then totalfee=fee*1.06;
MPRINT(CHOICE): else if location='Seattle' then totalfee=fee*1.025;
MPRINT(CHOICE): else if location='Dallas' then totalfee=fee*1.05;
MPRINT(CHOICE): run;
```

Earlier you learned the process that occurs when a macro program is compiled. Now that you have seen more-complex macro programs, we can examine this process again.

Remember that during macro compilation, macro statements are checked for syntax errors. If a macro definition contains macro statement syntax errors, error messages are written to the SAS log, and a non-executable (dummy) macro is created.

Example

Suppose you attempt to compile a macro that contains a syntax error. For example, the following program is missing a percent sign in the %IF-%THEN statement:

```
%macro printit;
   %if &syslast ne _NULL_ then %do;
      proc print data=_last_(obs=5);
      title "Last Created Data Set Is &syslast";
      run;
   %end;
%mend;
```

When you submit this macro definition, the macro processor checks the %IF-%THEN statement and the %DO and %END

statements for syntax errors. Since there is a syntax error in the %IF-%THEN statement, the following error messages are written to the SAS log.

Table 11.14: SAS Log

```
10     %macro printit;
11       %if &syslast ne _NULL_ then %do;
ERROR:   Macro keyword DO appears as text. A semicolon or other
         delimiter may be missing.
ERROR:   Expected %THEN statement not found. A dummy macro will be
         compiled.
12           proc print data=_last_(obs=5);
13           title "Last Created Data Set Is &syslast";
14           run;
15       %end;
ERROR:   There is no matching %DO statement for the %END. This
       statement will be ignored.
16 %mend;
```

Macro Execution with Conditional Processing

Earlier you learned that when the macro processor receives **`%macro-name`**, it executes compiled macro language statements such as %IF-%THEN. The values of macro variables that are used within the %IF logical expression are resolved during macro execution. The %IF logical expression is automatically evaluated.

Example

Suppose the *Printit* macro has been compiled and has been stored in the *Work.Sasmacr* catalog.

1. First, you submit a call to the *Printit* macro, as follows:

   ```
   %printit
   ```

2. The macro processor locates the macro in the SAS catalog *Work.Sasmacr*. Catalog Entry *Work.Sasmacr.Printit.Macro*

   ```
   %macro printit;
      %if &syslast ne _NULL_ %then %do;
         proc print data=_last_(obs=5);
         title "Last Created Data Set Is &syslast";
         run;
      %end;
   %mend;
   ```

3. The macro processor begins to execute compiled macro language statements from *Printit* (that is, the %IF-%THEN statement). Because the %IF expression is true, the %DO block is processed.

4. The macro processor places the text that follows the %DO statement (that is, the PROC PRINT step) on the input stack.

   Input Stack

   ```
   proc print data=_last_(obs=5);
      title "Last Created Data Set Is &syslast";
   run;
   ```

5. Word scanning proceeds as usual on the PROC PRINT step. When a macro trigger such as **`&syslast`** is encountered, the macro reference is passed to the macro processor for resolution. The macro processor returns resolved values to the input stack.

6. After the word scanner sends all of the tokens from the PROC PRINT step to the compiler, and the RUN statement is encountered, the PROC PRINT step executes.

7. Macro execution pauses while the PROC PRINT step executes, and macro execution stops when the %MEND statement is encountered.

It is possible to conditionally insert individual statements into the input stack, even in the middle of a step.

Example

Suppose you want to generate a report of enrollment at each training center as listed in the data set *Sasuser.All*. You can specify your macro program so that if a specific course is requested, the macro will insert a WHERE ALSO statement in order to restrict the report to that course. This example also customizes the second title line based on whether a course was selected, as follows:

```
%macro attend(crs,start=01jan2001,stop=31dec2001);
   %let start=%upcase(&start);
   %let stop=%upcase(&stop);
   proc freq data=sasuser.all;
      where begin_date between "&start"d and "&stop"d;
      table location / nocum;
      title "Enrollment from &start to &stop";
      %if &crs= %then %do;
         title2 "for all Courses";
      %end;
      %else %do;
         title2 "for Course &crs only";
         where also course_code="&crs";
      %end;
   run;
%mend;
```

Note In the program above, the %IF statement **`%if &crs=`** is true when **`crs`** has a value of null.

Suppose you submit the following call, which specifies a specific course:

```
%attend(C003)
```

This call results in the following output. Notice that the second title has been written according to the %ELSE %DO statement in the macro.

Enrollment from 01JAN2001 to 31DEC2001 for Course C003 only

The FREQ Procedure

Location

Location	Frequency	Percent
Boston	20	40.00
Seattle	30	60.00

Table 11.15: SAS Log

```
18     %attend(C003)
MPRINT(ATTEND):   proc freq data=sasuser.all;
MPRINT(ATTEND):   where begin_date between "01JAN2001"d and "31DEC2001"d;
MPRINT(ATTEND):   table location / nocum;
MPRINT(ATTEND):   title "Enrollment from 01JAN2001 to 31DEC2001";
MPRINT(ATTEND):   title2 "for Course C003 only";
MPRINT(ATTEND):   where also course_code="C003";
NOTE: WHERE clause has been augmented.
MPRINT(ATTEND):   run;

NOTE: Writing HTML Body file: sashtml.htm
NOTE: There were 2 observations read from the data set SASUSER.SCHEDULE.
WHERE (Course_Code='C003') and (Begin_Date>='01JAN2001'D and
                                        Begin_Date<='31DEC2001'D);
NOTE: There were 207 observations read from the data set SASUSER.STUDENTS.
NOTE: There were 434 observations read from the data set SASUSER.REGISTER.
NOTE: There were 1 observations read from the data set SASUSER.COURSES.
      WHERE Course_Code='C003';
NOTE: There were 50 observations read from the data set SASUSER.ALL.
WHERE (begin_date>='01JAN2001'D and begin_date<='31DEC2001'D)
                                          and (course_code='C003');
```

Now suppose you submit the following call, which specifies a start date but does not specify a course:

```
%attend(start=01jul2001)
```

This call results in the following output. Notice that in this output, the second title line is written according to the %IF-% THEN statement in the macro.

Enrollment from 01JUL2001 to 31DEC2001 for all Course

The FREQ Procedure

Location

Location	Frequency	Percent
Boston	56	34.57
Dallas	23	14.20
Seattle	83	51.23

Table 11.16: SAS Log

```
19    %attend(start=01jul2001)
MPRINT(ATTEND):    options mprint;
MPRINT(ATTEND):    proc freq data=sasuser.all;
MPRINT(ATTEND):    where begin_date between "01JUL2001"d and "31DEC2001"d;
MPRINT(ATTEND):    table location / nocum;
MPRINT(ATTEND):    title "Enrollment from 01JUL2001 to 31DEC2001";
MPRINT(ATTEND):    title2 "for all Courses";
MPRINT(ATTEND):    run;

NOTE: There were 6 observations read from the data set SASUSER.SCHEDULE.
      WHERE Course_Code in ('C001', 'C002', 'C003', 'C004', 'C005', 'C006') and
      (Begin_Date>='01JUL2001'D and Begin_Date<='31DEC2001'D);
NOTE: There were 207 observations read from the data set SASUSER.STUDENTS.
NOTE: There were 434 observations read from the data set SASUSER.REGISTER.
NOTE: There were 6 observations read from the data set SASUSER.COURSES.
NOTE: There were 162 observations read from the data set SASUSER.ALL.
      WHERE (begin_date>='01JUL2001'D and begin_date<='31DEC2001'D);
```

Conditional Processing of Parts of Statements

The text that is processed as the result of conditional logic can be a small part of a SAS statement. This makes it possible to conditionally insert text into the middle of a statement.

Example

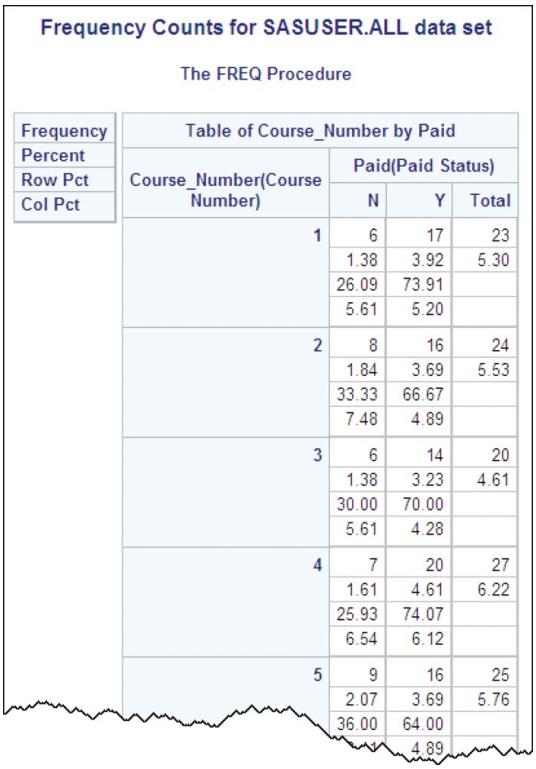
Suppose you want to print a table of frequency counts from a SAS data set. You can generate either a one-way table or a two-way table, based on the value of a macro parameter. This example creates a one-way table if only the **cols** parameter is specified in the call. It creates a two-way table if the **rows** parameter is also specified.

```
%macro counts (cols=_all_,rows=,dsn=&syslast);
   title "Frequency Counts for %upcase(&dsn) data set";
   proc freq data=&dsn;
      tables
      %if &rows ne %then &rows *;
      &cols;
   run;
%mend counts;
```

Suppose you submit the following call, which specifies both **cols** and **rows**:

```
%counts(dsn=sasuser.all, cols=paid, rows=course_number)
```

Part of the resulting output from this call is shown below. Notice that the macro has created a two-way table.



Frequency Counts for SASUSER.ALL data set

The FREQ Procedure

Frequency Percent Row Pct Col Pct	Table of Course_Number by Paid		
Course_Number(Course Number)	Paid(Paid Status)		
	N	Y	Total
1	6	17	23
	1.38	3.92	5.30
	26.09	73.91	
	5.61	5.20	
2	8	16	24
	1.84	3.69	5.53
	33.33	66.67	
	7.48	4.89	
3	6	14	20
	1.38	3.23	4.61
	30.00	70.00	
	5.61	4.28	
4	7	20	27
	1.61	4.61	6.22
	25.93	74.07	
	6.54	6.12	
5	9	16	25
	2.07	3.69	5.76
	36.00	64.00	
	8.41	4.89	

The log shows the generated PROC FREQ code from this macro.

Table 11.17: SAS Log

```
28   %counts(dsn=sasuser.all, cols=paid, rows=course_number)
MPRINT(COUNTS):   title "Frequency Counts for SASUSER.ALL data set";
MPRINT(COUNTS):   proc freq data=sasuser.all;
MPRINT(COUNTS):   tables course_number * paid;
MPRINT(COUNTS):   run;

NOTE: There were 18 observations read from the data set SASUSER.SCHEDULE.
      WHERE Course_Code in ('C001', 'C002', 'C003', 'C004', 'C005', 'C006');
NOTE: There were 207 observations read from the data set SASUSER.STUDENTS.
NOTE: There were 434 observations read from the data set SASUSER.REGISTER.
NOTE: There were 6 observations read from the data set SASUSER.COURSES.
NOTE: There were 434 observations read from the data set SASUSER.ALL.
NOTE: PROCEDURE FREQ used (Total process time):
```

Now suppose you submit the following call, which specifies **`cols`** but does not specify **`rows`:**

```
%counts(dsn=sasuser.all, cols=paid)
```

The output that results from this call is shown below. Notice that this time the macro has created a one-way table.

Frequency Counts for SASUSER.ALL data set

The FREQ Procedure

Paid Status

Paid	Frequency	Percent	Cumulative Frequency	Cumulative Percent
N	107	24.65	107	24.65
Y	327	75.35	434	100.00

The log shows the generated TABLES statement.

Table 11.18: SAS Log

```
29   %counts(dsn=sasuser.all, cols=paid)
MPRINT(COUNTS):    title "Frequency Counts for SASUSER.ALL data set";
MPRINT(COUNTS):    proc freq data=sasuser.all;
MPRINT(COUNTS):    tables paid;
MPRINT(COUNTS):    run;

NOTE: There were 18 observations read from the data set SASUSER.SCHEDULE.
      WHERE Course_Code in ('C001', 'C002', 'C003', 'C004', 'C005', 'C006');
NOTE: There were 207 observations read from the data set SASUSER.STUDENTS.
NOTE: There were 434 observations read from the data set SASUSER.REGISTER.
NOTE: There were 6 observations read from the data set SASUSER.COURSES.
NOTE: There were 434 observations read from the data set SASUSER.ALL.
NOTE: PROCEDURE FREQ used (Total process time):
```

Case Sensitivity in Macro Comparisons

Remember that comparisons that are made in %IF expressions are *case sensitive*.

Example

If you construct your %IF statement using incorrect case in any program text, the statement will *never* be true. For example, the %IF statement below will always be false because **_null_** is specified in lowercase but is always stored in SAS in uppercase:

```
%macro prtlast;
   %if &syslast=_null_ %then %do;
      %put No data sets created yet.;
   %end;
   %else %do;
      proc print;
         title "Last Created Data Set is &syslast";
      run;
   %end;
%mend;
options mprint mlogic symbolgen;
%prtlast
```

Suppose **SYSLAST** has a value of *_NULL_* when you submit this example. The following messages are written to the SAS log.

Table 11.19: SAS Log

```
29   %prtlast
MLOGIC(PRTLAST): Beginning execution.
SYMBOLGEN: Macro variable SYSLAST resolves to _NULL_
MLOGIC(PRTLAST): %IF condition &syslast = _null_ is FALSE

NOTE: The SAS System stopped processing this step because of errors.
NOTE: PROCEDURE PRINT used:
      real time           1:32.58
      cpu time            0.05 seconds

MPRINT(PRTLAST): proc print;
ERROR: There is not a default input data set (_LAST_ is _NULL_).
SYMBOLGEN: Macro variable SYSLAST resolves to _NULL_
MPRINT(PRTLAST): title "Last Created Data Set is _NULL_";
MPRINT(PRTLAST): run;

NOTE: The SAS System stopped processing this step because of errors.
NOTE: PROCEDURE PRINT used:
      real time           0.01 seconds
      cpu time            0.01 seconds
MLOGIC(PRTLAST): Ending execution.
```

Tip The %UPCASE function is often useful when you construct %IF statements. For more information about the %

UPCASE function, see "Introducing Macro Variables" on page 304.

Processing Statements Iteratively

Overview

Many macro applications require iterative processing. With the iterative %DO statement you can repeatedly

- execute macro programming code
- generate SAS code.

General form, iterative %DO statement with %END statement:

```
%DO index-variable=start %TO stop <%BYincrement>;
          text
%END;
```

where

index-variable

is either the name of a macro variable or a text expression that generates a macro variable name.

start and *stop*

specify either integers or macro expressions that generate integers to control how many times the portion of the macro between the iterative %DO and %END statements is processed.

increment

specifies either an integer (other than 0) or a macro expression that generates an integer to be added to the value of the index variable in each iteration of the loop. By default, *increment* is 1.

text

can be

- constant text, possibly including SAS data set names, SAS variable names, or SAS statements
- macro variables, macro functions, or macro program statements
- any combination of the above.

%DO and %END statements are valid *only inside* a macro definition. The *index-variable* is created in the local symbol table if it does not appear in any existing symbol table.

The iterative %DO statement evaluates the value of the index variable at the *beginning* of each loop iteration. The loop stops processing when the index variable has a value that is outside the range of the *start* and *stop* values.

Example

You can use a macro loop to create and display a series of macro variables.

This example creates a series of macro variables named **teachl-teachn**, one for each observation in the *Sasuser.Schedule* data set, and assigns teacher names to them as values. Then the *Putloop* macro uses a %DO statement and a %END statement to create a loop that writes these macro variables and their values to the SAS log, as follows:

```
data _null_;
   set sasuser.schedule end=no_more;
   call symput('teach'||left(_n_),(trim(teacher)));
   if no_more then call symput('count',_n_);
```

```
run;

%macro putloop;
   %local i;
   %do i=1 %to &count;
      %put TEACH&i is &&teach&i;
   %end;
%mend putloop;

%putloop
```

Tip It is a good idea to specifically declare the index variable of a macro loop as a local variable to avoid the possibility of accidentally changing the value of a macro variable that has the same name in other symbol tables.

When the *Putloop* macro is executed, no code is sent to the compiler, because the %PUT statements are executed by the macro processor. The following messages are written to the SAS log.

Table 11.20: SAS Log

```
TEACH1 is Hallis, Dr. George
TEACH2 is Wickam, Dr. Alice
TEACH3 is Forest, Mr. Peter
TEACH4 is Tally, Ms. Julia
TEACH5 is Hallis, Dr. George
TEACH6 is Berthan, Ms. Judy
TEACH7 is Hallis, Dr. George
TEACH8 is Wickam, Dr. Alice
TEACH9 is Forest, Mr. Peter
TEACH10 is Tally, Ms. Julia
TEACH11 is Tally, Ms. Julia
TEACH12 is Berthan, Ms. Judy
TEACH13 is Hallis, Dr. George
TEACH14 is Wickam, Dr. Alice
TEACH15 is Forest, Mr. Peter
TEACH16 is Tally, Ms. Julia
TEACH17 is Hallis, Dr. George
TEACH18 is Berthan, Ms. Judy
```

You can also use a macro loop to generate statements that can be placed inside a SAS program step.

Example

The following macro generates a series of statements within a DATA step. On each iteration, the macro writes a message to the SAS log that puts the current value of the index variable into HEX6. format.

```
%macro hex(start=1,stop=10,incr=1);
   %local i;
   data _null_;
      %do i=&start %to &stop %by &incr;
         value=&i;
         put "Hexadecimal form of &i is " value hex6.;
      %end;
   run;
%mend hex;
```

Note The HEX6. format converts a number to hexadecimal format.

Suppose you submit the following call:

```
options mprint mlogic;
%hex(start=20,stop=30,incr=2)
```

Some of the messages that are written to the SAS log when *Hex* executes are shown below. Notice that according to the MLOGIC messages, the loop stops processing when the value of the index variable is *32* (which is beyond the value that is specified for **`Stop`)**.

Table 11.21: SAS Log

```
MLOGIC(HEX): %DO loop index variable I is now 30; loop will
               iterate again.
MPRINT(HEX): value=30;
MPRINT(HEX): put "Hexadecimal form of 30 is " value hex6.;
MLOGIC(HEX): %DO loop index variable I is now 32; loop will
               not iterate again.
MPRINT(HEX): run;

Hexadecimal form of 20 is 000014
Hexadecimal form of 22 is 000016
Hexadecimal form of 24 is 000018
Hexadecimal form of 26 is 00001A
Hexadecimal form of 28 is 00001C
Hexadecimal form of 30 is 00001E
NOTE: DATA statement used:
      real time          0.06 seconds
      cpu time           0.06 seconds

MLOGIC(HEX): Ending execution.
```

Generating Complete Steps

You can use the iterative %DO statement to build macro loops that create complete SAS steps.

Example

Suppose course offerings for several years are stored in a series of external files that are named by year, such as *Rawl999.dat* and *Raw2000.dat*. All the files have the same record layout. Suppose you want to read each file into a separate SAS data set.

The following macro uses a %DO statement to create a loop that creates a data set from each of the specified external files:

```
%macro readraw(first=1999,last=2005);
   %local year;
   %do year=&first %to &last;
      data year&year;
         infile "raw&year..dat";
         input course_code $4.
         location $15.
         begin_date date9.
         teacher $25.;
       run;

       proc print data=year&year;
          title "Scheduled classes for &year";
          format begin_date date9.;
       run;
   %end;
%mend readraw;
```

Suppose you submit the following call to the *Readraw* macro:

```
%readraw(first=2000,last=2002)
```

The macro creates three data sets named *Year2000, Year2001*, and *Year2002*. Remember that in order for this program to run properly, the raw data files must be named appropriately, and they must be stored in the location that the program specifies. The generated DATA step is shown in the log.

Table 11.22: SAS Log

```
336 %readraw(first=2000,last=2002)
MLOGIC(READRAW): Beginning execution.
MLOGIC(READRAW): Parameter FIRST has value 2000
MLOGIC(READRAW): Parameter LAST has value 2002
MLOGIC(READRAW): %LOCAL YEAR
MLOGIC(READRAW): %DO loop beginning; index variable YEAR; start value is 2000;
```

```
     stop value is 2002; by value is 1.
MPRINT(READRAW):    data year2000;
MPRINT(READRAW):    infile "raw2000.dat";
MPRINT(READRAW):    input course_code $4. location $15. begin_date date9.
                          teacher $25.;
MPRINT(READRAW): run;
```

Using Arithmetic and Logical Expressions

The %EVAL Function

The %EVAL function evaluates integer arithmetic or logical expressions. Logical expressions and arithmetic expressions are sequences of operators and operands forming sets of instructions that are evaluated to produce a result.

- An arithmetic expression contains an arithmetic operator.
- A logical expression contains a logical operator.

General form, %EVAL function:

%EVAL(*arithmetic or logical expression)*

The %EVAL function

- translates integer strings and hexadecimal strings to integers
- translates tokens representing arithmetic, comparison, and logical operators to macro-level operators
- performs arithmetic and logical operations.

For arithmetic expressions, if an operation results in a non-integer value, %EVAL truncates the value to an integer. Also, %EVAL returns a null value and issues an error message when non-integer values are used in arithmetic expressions.

%EVAL evaluates logical expressions and returns a value to indicate if the expression is true or false. A value of 0 indicates that the expression is false, and a value of 1 or any other numeric value indicates that the expression is true.

The %EVAL function *does not* convert the following to numeric values:

- numeric strings that contain a period or E-notation
- SAS date and time constants.

Following are some examples.

Examples

The following table shows several examples of arithmetic and logical expressions, as well as the results that %EVAL produces when it evaluates these expressions.

If you submit these statements…	These messages are written to the log…
`%put value=%eval(10 lt 2);`	`value=0`
`%put value=10+2;` `%put value=%eval(10+2);`	`value=10+2` `value=12`

`%let counter=2;`
`%let counter=%eval(&counter+1);`
`%put counter=&counter;`
	`counter=3`

`%let numer=2;`
`%let denom=8;`
`%put value=%eval(&numer/&denom);`
	`value=0`

`%let numer=2;`
`%let demon=8;`
`%put value=%eval(&numer/&denom*&denom);`
`%put value=%eval(&denom*&numer/&denom);`
	`value=0`
	`value=2`

`%let real=2.4;`
`%let int=8;`
`%put value=%eval(&real+&int);`
	`value=`

In the last example above, the decimal value of the **`real`** variable causes an error message to be written to the SAS log, as shown here.

Table 11.23: SAS Log

```
1    %let real=2.4;
2    %let int=8;
3    %put value=%eval(&real+&int);
ERROR: A character operand was found in the %EVAL function
or %IF condition where a numeric operand is required.
The condition was: 2.4+8
value=
```

Because %EVAL does not convert a value that contains a period to a number, the operands are evaluated as character operands.

You have seen that the %EVAL function generates ERROR messages in the log when it encounters an expression that contains non-integer values. In order to avoid these ERROR messages, you can use the *%SYSEVALFfunction*. The %SYSEVALF function evaluates arithmetic and logical expressions using floating-point arithmetic.

General form, %SYSEVALF function:

%SYSEVALF*(expression<, conversion-type>)*

where

expression

is an arithmetic or logical expression to evaluate.

conversion-type

optionally converts the value returned by %SYSEVALF to the type of value specified. Conversion-type can be BOOLEAN, CEIL, FLOOR, or INTEGER.

The %SYSEVALF function performs floating-point arithmetic and returns a value that is formatted using the BEST16. format. The result of the evaluation is always text.

Example

The macro in the following example performs all types of conversions for values in the %SYSEVALF function:

```
%macro figureit(a,b);
   %let y=%sysevalf(&a+&b);
   %put The result with SYSEVALF is: &y;
   %put BOOLEAN conversion: %sysevalf(&a +&b, boolean);
   %put CEIL conversion: %sysevalf(&a +&b, ceil);
   %put FLOOR conversion: %sysevalf(&a +&b, floor);
```

```
   %put INTEGER conversion: %sysevalf(&a +&b, integer);
   %mend figureit;

%figureit(100,1.59)
```

Executing this program writes the following lines to the SAS log.

Table 11.24: SAS Log

```
The result with SYSEVALF is: 101.59
BOOLEAN conversion: 1
CEIL conversion: 102
FLOOR conversion: 101
INTEGER conversion: 101
```

Automatic Evaluation

%SYSEVALF is the *only* macro function that can evaluate logical expressions that contain floating point, date, time, datetime, or missing values. Specifying a conversion type can prevent problems when %SYSEVALF returns missing or floating-point values to macro expressions or macro variables that are used in other macro expressions that require an integer value.

Keep in mind that any macro language function or statement that requires a numeric or logical expression automatically invokes the %EVAL function. This includes the %SCAN function, the %SUBSTR function, the %IF-%THEN statement, and more.

Summary

Contents

This section contains the following topics.

Text Summary

Basic Concepts

A macro program is created with a macro definition, which consists of a %MACRO statement and a %MEND statement. The %MACRO statement also provides a name for the macro. Any combination of macro language statements and SAS language statements can be placed in a macro definition. The macro definition must be compiled before it is available for execution. The MCOMPILENOTE= option will cause a note to be issued to the SAS log when a macro has completed compilation. To execute a name style macro, you submit a call to the macro by preceding the macro name with a percent sign.

Developing and Debugging Macros

Two system options, MLOGIC and MPRINT, are useful for macro development and debugging. The MLOGIC option writes messages that trace macro execution to the SAS log. The MPRINT option prints the text that is sent to the compiler after all macro resolution has taken place. The SYMBOLGEN option and macro comments are also useful for macro development and debugging.

Using Macro Parameters

You can use parameter lists in your macro definition in order to make your macros more flexible and easier to adapt. Parameters can be either positional or keyword. You can also use mixed parameter lists that contain both positional and keyword parameters. Parameters define macro variables that can take on different values when you call the macro, including null values. You can use the PARMBUFF option in conjunction with the automatic macro variable **SYSPBUFF** to

define a macro that accepts a varying number of parameters each time you call it.

Understanding Symbol Tables

When a macro executes, it sometimes creates its own temporary symbol table, called a local symbol table. The local symbol table exists in addition to the global symbol table. If a macro creates or resolves macro variables, a local symbol table might be used. In order to fully control macro behavior, you must understand the basic rules that the macro processor uses to determine which symbol table to access under specific circumstances. Statements such as %GLOBAL and %LOCAL enable you to explicitly define where macro variables are stored. The %SYMDEL statement enables you to delete a macro variable from the global symbol table during a SAS session.

You can call a macro within a macro definition. That is, you can nest macros. When a nested macro is called, multiple local symbol tables can exist. The MPRINTNEST and MLOGICNEST options provide nesting information in the messages that are written to the SAS log for the MPRINT and MLOGIC options.

Processing Statements Conditionally

Conditional processing is available with the %IF-%THEN/%ELSE statements. These statements control what action the macro processor takes when an expression evaluates to true or to false. The action could be the execution of other macro programming statements or the placement of text onto the input stack. If the code that is used to describe this action includes multiple statements, you must enclose this code between a %DO statement and a %END statement.

It is possible to conditionally place whole SAS steps, whole SAS statements, or parts of SAS statements onto the input stack.

Processing Statements Iteratively

To perform repetitive actions, you can use %DO loops. You can use iterative processing to generate complete SAS steps, individual statements, or data-dependent steps.

Using Arithmetic and Logical Expressions

You use the %EVAL function to evaluate arithmetic or logical expressions that do not contain any non-integer or missing values. Macro language functions and statements that require a numeric or logical expression automatically use the %EVAL function. You use the %SYSEVALF function to evaluate arithmetic or logical expressions that contain non-integer or missing values.

Syntax

```
%MACRO macro-name;
   text
%MEND <macro-name>;
OPTIONS MCOMPILENOTE= NONE | NOAUTOCALL | ALL;
OPTIONS MPRINT | NOPRINT;
OPTIONS MLOGIC | NOMLOGIC;
OPTIONS MLOGICNEST | NOMLOGICNEST;
OPTIONS MPRINTNEST | NOMPRINTNEST;
%*comment;
%MACRO macro-name(parameter-1<,…,parameter-n>);
   text
%MEND <macro-name>;
%MACRO macro-warne/PARMBUFF;
   text
%MEND;
%MACRO macro-name(keyword-1=<value-1>
         <,…,keyword-n=<value-n>>);
   text
%MEND <macro-name>;
%MACRO macro-name(parameter-1<,…,parameter-n>,
         keyword-1=<value-1 ><,…,keyword-n=<value-n>>);
   text
%MEND <macro-name>;
%GLOBAL macro-variable-1 <…macro-variable-n>;
%LOCAL macro-variable-1 <…macro-variable-n>;
%IF expression %THEN text;
<%ELSEtext;>
```

```
%IF expression %THEN %DO;
         text and/or macro language statements
%END;
%ELSE %DO;
      text and/or macro language statements
%END;
%DO index-variable=start %TO stop %BY increment;
        text
%END;
%EVAL(arithmetic or logical expression)
%SYSEVALF(expression<, conversion-type>)
```

Sample Programs

Defining a Basic Macro

```
%macro prtlast;
   proc print data=&syslast (obs=5);
   title "Listing of &syslast data set";
   run;
%mend;
```

Defining a Macro with Positional Parameters

```
%macro printdsn(dsn,vars);
   proc print data=&dsn;
      var &vars;
      title "Listing of %upcase(&dsn) data set";
   run;
%mend;
```

Defining a Macro with Keyword Parameters

```
%macro printdsn(dsn=sasuser.courses,
              vars=course_code
              course_title days);
   proc print data=&dsn;
      var &vars;
      title "Listing of %upcase(&dsn) data set";
   run;
%mend;
```

Defining a Macro with Mixed Parameters

```
%macro printdsn(dsn, vars=course_title course_code days);
   proc print data=&dsn;
      var &vars;
      title "Listing of %upcase(&dsn) data set";
   run;
%mend;
```

Using the %IF-%THEN Statement

```
%macro choice(status);
   data fees;
      set sasuser.all;
      %if &status=PAID %then %do;
         where paid='Y';
         keep student_name course_code begin_date totalfee;
      %end;
      %else %do;
         where paid='N';
         keep student_name course_code
              begin_date totalfee latechg;
         latechg=fee*1.10;
      %end;
      /* add local surcharge */
     if location='Boston' then totalfee=fee*1.06;
     else if location='Seattle' then totalfee=fee*1.025;
     else if location='Dallas' then totalfee=fee*1.05;
   run;
```

```
%mend choice;
```

Using the Iterative %DO Statement

```
%macro hex(start=1,stop=10,incr=1);
   %local i;
   data _null_;
   %do i=&start %to &stop %by &incr;
      value=&i;
      put "Hexadecimal form of &i is " value hex6.;
   %end;
   run;
%mend hex;
options mprint mlogic symbolgen;
%hex(start=20,stop=30,incr=2)
```

Points to Remember

- Macro programs are defined by using a %MACRO statement and a %MEND statement.
- Macros are executed and called by typing a % before the name of the macro.
- The MPRINT, MLOGIC, and SYMBOLGEN system options can be useful for developing and debugging macro programs.
- Parameters can make your macro programs more flexible by creating local macro variables whose values can be updated by the macro call.
- You can use the %IF-%THEN statement to conditionally process whole SAS steps, SAS statements, or parts of statements.
- You can use the iterative %DO statement to create macro loops that can process repetitive tasks.

Quiz

Select the best answer for each question. After completing the quiz, check your answers using the answer key in the appendix.

1. Which of the following is false? ?

a. A %MACRO statement must always be paired with a %MEND statement.

b. A macro definition can include macro variable references, but it cannot include SAS language statements.

c. Only macro language statements are checked for syntax errors when the macro is compiled.

d. Compiled macros are stored in a temporary SAS catalog by default.

2. Which of the following examples correctly defines a macro named *Print* that implements parameters named **`vars`** and **`total`**? ?

a.
```
a.  %macro print(vars, total);
     proc print data=classes;
        var vars;
        sum total;
     run;
     %mend print;
```

b.
```
b.  %macro print('vars', 'total');
     proc print data=classes;
        var &vars;
        sum &total;
     run;
     %mend print;
```

c.
```
c.  %macro print(vars, total);
```

```
        proc print data=classes;
           var &vars;
           sum &total;
        run;
      %mend print;
```

d. d.
```
%macro print(vars, total);
  proc print data=classes;
     var :vars;
     sum :total;
  run;
 %mend print;
```

3. Which of the following correctly references the macro named *Printdsn* as shown here: ?

```
%macro printdsn(dsn,vars);
   %if &vars= %then %do;
      proc print data=&dsn;
      title "Full Listing of %upcase(&dsn) data set";
      run;
  %end;
  %else %do;
     proc print data=&dsn;
        var &vars;
     title "Listing of %upcase(&dsn) data set";
     run;
  %end;
%mend;
```

a. a. `%printdsn(sasuser.courses, course_title days);`

b. b. `%printdsn(dsn=sasuser.courses, vars=course_title days)`

c. c. `%printdsn(sasuser.courses, course_title days)`

d. d. `%printdsn(sasuser.courses, course_title, days)`

4. If you use a mixed parameter list in your macro program definition, which of the following is false? ?

a. You must list positional parameters before any keyword parameters.

b. Values for both positional and keyword parameters are stored in a local symbol table.

c. Default values for keyword parameters are the values that are assigned in the macro definition, whereas positional parameters have a default value of null.

d. You can assign a null value to a keyword parameter in a call to the macro by omitting the parameter from the call.

5. Which of the following is false? ?

a. A macro program is compiled when you submit the macro definition.

b. A macro program is executed when you call it **(`%macro-name`)**.

c. A macro program is stored in a SAS catalog entry only after it is executed.

d. A macro program is available for execution throughout the SAS session in which it is compiled.

6. When you use an %IF-%THEN statement in your macro program, ?

a. you must place %DO and %END statements around code that describes the conditional action, if that code contains multiple statements.

b. the %ELSE statement is optional.

c. you cannot refer to DATA step variables in the logical expression of the %IF statement.

d. allof the above.

7. Which of the following can be placed onto the input stack? ?

a. only whole steps.

b. only whole steps or whole statements.

c. only whole statements or pieces of text within a statement.

d. whole steps, whole statements, or pieces of text within statements.

8. Which of the following will create a macro variable named **`class`** in a local symbol table? ?

a.
```
a. data _null_;
    set sasuser.courses;
    %let class=course_title;
   run;
```

b.
```
b. data _null_;
    set sasuser.courses;
    call symput('class', course_title);
   run;
```

c.
```
c. %macro sample(dsn);
    %local class;
    %let class=course_title;
    data_null_;
       set &dsn;
    run;
   %mend;
```

d.
```
d. %global class;
   %macro sample(dsn);
     %let class=course_title;
     data _null_;
        set &dsn;
     run;
   %mend;
```

9. Which of the following examples correctly defines the macro program *Hex?* ?

a.
```
a. %macro hex(start=1, stop=10, incr=1);
     %local i;
     data _null_;
     %do i=&start to &stop by &incr;
        value=&i;
        put "Hexadecimal form of &i is " value hex6.;
     %end;
     run;
   %mend hex;
```

b.
```
b. %macro hex(start=1, stop=10, incr=1);
     %local i;
     data _null_;
     %do i=&start %to &stop %by &incr;
        value=&i;
        put "Hexadecimal form of &i is " value hex6.;
     %end;
     run;
   %mend hex;
```

c.
```
c. %macro hex(start=1, stop=10, incr=1);
     %local i;
```

```
      data _null_;
      %do i=&start to &stop by &incr;
         value=&i;
         put "Hexadecimal form of &i is " value hex6.;
      run;
   %mend hex;
```

d. d.
```
   %macro hex(start=1, stop=10, incr=1);
      %local i;
      data _null_;
      %do i=&start to &stop by &incr;
         value=&i;
         put "Hexadecimal form of &i is " value hex6.;
      %end
      run;
   %mend hex;
```

10. When you submit a call to a compiled macro, what happens? ?

a. First, the macro processor checks all macro programming statements in the macro for syntax errors.

   Then the macro processor executes all statements in the macro.

b. The macro processor executes compiled macro programming statements.

   Then any SAS programming language statements are executed by the macro processor.

c. First, all compiled macro programming statements are executed by the macro processor.

   After all macro statements have been processed, any SAS language statements are passed back to the input stack in order to be passed to the compiler and then executed.

d. The macro processor executes compiled macro statements.

   If any SAS language statements are encountered, they are passed back to the input stack.

   The macro processor pauses while those statements are passed to the compiler and then executed.

   Then the macro processor continues to repeat these steps until it reaches the %MEND statement.

Answers

1. Correct answer: b

A macro definition must begin with a %MACRO statement and must end with a %MEND statement. The macro definition can include macro language statements as well as SAS language statements. When the macro is compiled, macro language statements are checked for syntax errors. The compiled macro is stored in a temporary SAS catalog by default.

2. Correct answer: c

To include positional parameters in a macro definition, you list the parameters in parentheses and separate them with commas. When the macro is executed, macro variables will be created in the local symbol table and will have the same names as the parameters. You can then use these macro variables within the macro.

3. Correct answer: c

To call a macro that includes positional parameters, you precede the macro name with a percent sign. You list the values for the macro variables that are defined by the parameters in parentheses. List values in the same order in which the parameters are listed, and separate them with commas. Remember that a macro call is not a SAS language statement and does not require a semicolon.

4. Correct answer: d

In a mixed parameter list, positional parameters must be listed before any keyword parameters. Both positional and keyword parameters create macro variables in the local symbol table. To assign a null value to a keyword parameter, you list the parameter without a value in the macro call.

5. Correct answer: c

When you submit a macro definition, the macro is compiled and is stored in a SAS catalog. Then when you call the macro, the macro is executed. The macro is available for execution anytime throughout the current SAS session.

6. Correct answer: d

You can use %IF-%THEN statements to conditionally process code. Within a %IF-%THEN statement, you must use %DO and %END statements to enclose multiple statements. %IF-%THEN statements are similar to IF THEN statements in the DATA step, but they are part of the macro language.

7. Correct answer: d

By using %IF-%THEN statements, you can place whole steps, individual statements, or parts of statements onto the input stack.

8. Correct answer: c

There are several ways to create macro variables in the local symbol table. Macro variables that are created by parameters in a macro definition or by a %LOCAL statement are always created in the local table. Macro variables that are created by a %LET statement or by the SYMPUT routine inside a macro definition might be created in the local table as well.

9. Correct answer: b

To define macros with %DO loops you use a %DO statement and a %END statement. Be sure to precede all keywords in the statements with percent signs since the %DO and %END statements are macro language statements. Also, be sure to end these statements with semicolons.

10. Correct answer: d

When you submit a call to a compiled macro, the macro is executed. Specifically, the macro processor executes compiled macro language statements first. When any SAS language statements are encountered, the macro processor places these statements onto the input stack and pauses while they are passed to the compiler and then executed. Then the macro processor continues to repeat these steps until the %MEND statement is reached.